Nima Amir Dastmalchi (505320372)
CS 131 Homework 6

Problem 1 Part a

```
def foo(a):
   # a = 1
   a = 3 # a binds to value 3
   # copy value of a (3) into bar's first parameter.
   # copy result of baz() (5) into bar's second parameter
   bar(a, baz())
def bar(a, b):
   # a = 3, b = 5
   print("bar") # 2. print "bar"
   a = a + 1 # increment a by 1 to get 4
def baz():
   print("baz") # 1. print "baz"
   return 5
a = 1
foo(a) # copy value 1 into the parameter of foo
print(a) # 3. print "1"
```

Output:

baz
bar
1

Problem 1 Part b

```
def foo(a):
   # a refers to global variable a
   a = 3 # global variable a now holds 3
   bar(a, baz())
def bar(a, b):
   # a refers to global variable a (which currently holds 3)
   # b holds 5
   print("bar") # 2. print "bar"
   # increment global variable a to 4
   a = a + 1
def baz():
   print("baz") # 1. print "baz"
   return 5
a = 1
foo(a)
```

```
print(a) # 3. print "4"
```

Output:

bar
baz
4

Problem 1 Part c

```
def foo(a):
   # local var a points to object int(1)
   a = 3 # local var a points to object int(3)
   bar(a, baz())
def bar(a, b):
   # local var a points to object int(3)
   # local var b points to object int(5)
   print("bar") # 2. print "bar"
   a = a + 1 # local var a poitns to object int(4)
def baz():
   print("baz") # 1. print "baz"
   return 5
a = 1 # a points to object int(1)
foo(a)
print(a) # 3. print "1"
```

Output:

baz
bar
1

Problem 1 Part d

```
def foo(a):
   # a points to thunk(1)
   a = 3 # a now points to thunk(3)
   bar(a, baz())
def bar(a, b):
   # a points to thunk(3)
   # b points to thunk(baz())
   print("bar") # 1. print "bar"
   a = a + 1 # a points to thunk(thunk(3) + 1)
def baz():
   print("baz")
   return 5
```

```
a = 1 # a points to thunk(1)
foo(a)
print(a) # evaluate thunk(1), which is 1
# 2. print "1"
```

Output:

bar
1

Observe how the function baz is never executed because the value of b in the function bar is never referenced (and thus never evaluated). So, “baz” is never printed.

Problem 2

The Optional approach returns an object that represents either a single returned value or a generic (one-mode) failure. As such, the user of the function would have to check whether the returned Optional contains a result or an error (nullptr) via an if statement. For example, the user may use the function like this:

```
auto optional = firstIndexA(arr, size, n);
if (optional.value == nullptr) {
    // Handle failure
} else {
    int index = *optional.value;
    // Handle success
}
```

On the other hand, using C++’s native exception handling would mean that the function can throw an exception in case of an error (in this case, the only error is when the element is not in the array). Therefore, the user of the function will need to handle the exception using try-catch blocks.

```
try {
    int index = *firstIndexB(arr, size, n);
} catch(std::exception& e) {
    // Handle error
}
```

The Optional approach is more suitable for this use case because the function only has one mode of failure. The only error is when the element is not in the array, so we do not have an index to return. Therefore, we can just return an empty Optional. Otherwise, we can return an Optional that contains the index.

Note the C++ exception class hierarchy for Problem 3 (Source: https://stdcxx.apache.org/doc/stdlibug/18-2.html)

```
exception
   logic_error
      domain_error
      invalid_argument
      length_error
      out_of_range
   runtime_error
      range_error
      overflow_error
      underflow_error
```

Problem 3 Part a

Output of bar(0):

catch 2
I'm done!
that's what I say
Really done!

Problem 3 Part b

Output of bar(1):

catch 1
hurray!
I'm done!
that's what I say
Really done!

Problem 3 Part d

ch 3

For part d, I'm assuming that the 'return' statement will return from the function (not the catch block).

Problem 4 Part a

Concrete objects of an interface type cannot exist because interfaces are not concrete! An interface provides a set of unimplemented function definitions that the classes that implement the interface must implement. Thus, the class type can be used to create objects, but the interface type cannot because its functions are unimplemented. On the other hand, we can create object references/references of interface types that point/refer to a class type. Doing so guarantees (at compile-time) that the object that is pointed to by the object reference or referred to by the reference variable implements the interface functions.

Problem 4 Part b

No, the concept of interfaces in a dynamically typed language does not make sense. One of the main benefits of defining interfaces and having classes implemented from those interfaces is to create functions that operate on all classes that implement a particular interface. For example, we can have a function that can only operate on objects whose class implements a Comparable interface.

In a dynamically typed language, we can just pass our objects to a function that expects a particular method to be implemented. If that method is not implemented, then the language will throw a runtime error.

Problem 4 Part c

Use case #1: Elements of a sorted container must be comparable. That is, given two objects a, b, we must be able to compare a and b (to determine which one is smaller) to be able to insert them in the sorted container. This container will, therefore, only function on objects that implement functionality to compare themselves to another object of the same class. We can define an interface containing the signature of the comparison operator and have the container only accept objects that implement this interface.

Use case #2: It is desirable to separate the API from the implementation. For example, we may want to implement a list. We can implement this list as a linked list or a vector. The operations for both implementations are similar (access, insert, iterate, etc.). Thus, we can create an interface that defines these functions (access, insert, iterate, etc.) and both implementations can be inserted in classes that implement this interface. We can then write code that uses objects of the list interface and we can change the implementation of the list without needing to change much code.

Problem 7

Composition with delegation is a better approach.

The vector class implements functions for random access, insertion to any position, erasing from any position, etc. When the Stack class inherits from the vector class, we publicly inherit all these functions and the users of the Stack class also have access to these methods. This is bad because that's not how a Stack should be used. Instead, we can create a vector private instance in the Stack class and only push and pop from the vector from the same location. This would not expose the methods of vector class to users of Stack.

Problem 8 Part a

Consider the case where we have two classes A and B that both implement a "performAction" function with the same signature. Now, a class C inherits from both A and B without overriding the "performAction" method. It is ambiguous which implementation of the "performAction" method (A's or B's) the objects of class C should use when a user calls "performAction" on them.

On the other hand, if both A and B were interfaces and class C implemented both interfaces, there will be no problem because interfaces do not provide implementation. C must implement the "performAction" function because of the interfaces and there will be no ambiguity.

Problem 9 Part a

Static method binding:

1. Pros
    a. Speed: static method binding will not require us to look through a virtual table at runtime to determine which method to use for each object.
    b. Flexibility: We have the choice of using both dynamic and static method binding. We can use the 'virtual' keyword to make this choice.
2. Cons
    a. Bugs: It is more likely to write buggy code. We may forget to mark a superclass method as 'virtual' and later reimplement the function in a subclass. If we have object references/references of superclass type pointing to the subclass type, calling that method will use the base class implementation, which may not be the expected behavior.

Dynamic method binding

1. Pros
    a. Less buggy code (see reason above)
2. Cons
    a. Speed (see reason above)

Problem 9 Part b

The bark() method will be referenced in C++'s vtable because it is marked as 'virtual'.